A Work-Efficient Algorithm for Parallel Unordered Depth-First Search

Umut A. Acar, Arthur Charguéraud, Mike Rainey

A Work-Efficient Algorithm for Parallel Unordered Depth-First Search

Umut A. Acar
Carnegie Mellon University
and Inria
umut@cs.cmu.edu

Arthur Charguéraud
Inria & LRI Université
Paris Sud, CNRS
arthur.chargueraud@inria.fr

Mike Rainey
Inria
mike.rainey@inria.fr

ABSTRACT

Advances in processing power and memory technology have made multicore computers an important platform for high-performance graph-search (or graph-traversal) algorithms. Since the introduction of multicore, much progress has been made to improve parallel breadth-first search. However, less attention has been given to algorithms for unordered or loosely ordered traversals.

We present a parallel algorithm for unordered depth-first-search on graphs. We prove that the algorithm is work efficient in a realistic algorithmic model that accounts for important scheduling costs. This work-efficiency result applies to all graphs, including those with high diameter and high out-degree vertices. The algorithmic techniques behind this result include a new data structure for representing the frontier of vertices in depth-first search, a new amortization technique for controlling excess parallelism, and an adaptation of the lazy-splitting technique to depth first search.

We validate the theoretical results with an implementation and experiments. The experiments show that the algorithm performs well on a range of graphs and that it can lead to significant improvements over comparable algorithms.

1 Introduction

High-performance graph-search algorithms have become increasingly important in a variety of areas, such as social networks [31, 33, 49], physical sciences [2], and parallel garbage collection [22, 23, 47, 13]. Although there has been much research on parallel breadth-first search on directed graphs [39, 37, 10, 34] [8, 32, 44, 45, 50, 12, 19], and connectivity algorithms on undirected graphs [46, 36], other graph-search algorithms, such as parallel depth-first search on directed graphs, have received less attention.

One reason may be the difficulty of providing a parallel solution to the classic depth-first-search problem. For example, Reif [41], has shown that DFS is P-complete, suggesting that DFS is difficult to parallelize. There are solutions for certain special instances of the problem, for example, in planar graphs [24], but in the general case, DFS continues to be challenging: the best known algorithm is randomized and requires $O(\log^7(n))$ parallel time using $O(n^{2.376})$ processors, which is far from work efficient [9].

The primary difficulty in parallelizing DFS is the lexicographical ordering property that requires visiting the out-edges of a vertex in order. In many applications, such as reachability, graph search, and garbage collection [29], lexicographical ordering is not necessary. Prior work therefore considered *parallel unordered DFS* algorithms, which are sometimes called *Pseudo Parallel Depth First Search* or *PDFS* [20, 40, 38, 30]. For brevity, we refer to parallel unordered DFS or pseudo DFS as *PDFS*. Since they do not have to observe the edge ordering, PDFS algorithms can be asymptotically work efficient, performing $O(n + m)$ work, where n and m are the number of vertices and edges respectively, when ignoring load balancing and scheduling costs. However, when the cost of scheduling operations are included in the analysis, all known algorithms can incur large overheads. Our goal in this paper is to present a PDFS algorithm that operates with small overheads including the overheads of scheduling. We refer to such algorithms as *strongly work efficient*.

Including scheduling overheads as part of the work efficiency is important because such overheads are critical for performance. Cong et al [20] presented a PDFS algorithm whose purpose is to improve work efficiency on modern multicore computers. Their PDFS algorithm improves work efficiency by the application of a heuristic for adaptively batching edges. The idea behind the heuristic is to use the number of vertices owned by a processor as an estimate of total load in the system. They show empirically that their PDFS algorithm performs well on certain graphs, but do not show the algorithm to be strongly work-efficient.

In this paper, we present a strongly work-efficient PDFS algorithm. We prove that our algorithm achieves strong work efficiency by bounding important scheduling overheads, is implementable on modern CMPs, such as multicores, and performs well on a range of graphs, including graphs with high diameter and small amounts of parallelism. Our specific contributions include the following.

- A *frontier data structure* for representing the "frontier" in graph search. The data structure supports balanced splitting operations needed for effective parallel execution.
- A parallel thread-creation strategy that delivers a high degree of parallelism, while ensuring small overheads.
- A proof of strong work efficiency that is with respect to a careful specification of the algorithm in a realistic parallel model.
- Implementation and experimental evaluation that field-tests the algorithm and the theory, while evaluating the constant factors involved in the implementation and comparing with the state of the art.
- A modest additional empirical study of the locality of PDFS algorithm.

2 Overview

We present a high-level overview of our parallel unordered DFS (PDFS) algorithm in the context of the related work.

In a PDFS algorithm, each processor maintains a *frontier* of vertices. As in sequential DFS, a frontier stores the subset of visited vertices, whose outgoing edges have not yet been explored. Each processor works locally on its own frontier by repeatedly popping a vertex from its frontier and exploring its outgoing edges to discover new, unvisited vertices. When a processor discovers a new vertex, it attempts to visit the vertex by using an atomic read-modify-write operation (such as compare-and-swap) and, if it succeeds, adds the vertex to its frontier. By using an atomic operation, the algorithm ensures that each vertex is visited at most once.

To minimize the run time on a parallel machine, a PDFS algorithm needs to generate parallelism and perform load balancing to keep all the processors busy. A naive approach to this end would be to generate one thread for each vertex in the frontier. The threads can then be distributed over the processors by using a load balancing algorithm, such as work stealing [15], where an idle processor steals a thread (usually the "oldest" thread) from a (usually randomly chosen) busy processor, effectively redistributing work lazily and as needed.

The naive approach has two important limitations, which stem from the very fine granularity of work assigned to each thread. First, the cost of creating a thread for each vertex does not outweigh the benefits of parallelism. Second, since a thread contains only a single vertex, it may not generate significant work (the amount of work is proportional to the total number of vertices reachable from the vertex), causing large number of expensive load balancing (work-stealing) operations.

Cong et al [20] proposed a *batching* technique to ameliorate these problems. In their approach, each thread corresponds to a batch of vertices instead of a single vertex. Each batch is represented as a fixed-capacity buffer of vertices, (e.g. 128 vertices in a batch). Processors perform load balancing by stealing the oldest thread, and as such, are able to migrate a batch of work to another processor at each steal. The effect of this technique is to amortize thread creation and migration over the vertices contained in the batch.

By batching vertices into a single vertex, Cong et al's technique controls the overheads of parallelism by reducing the amount of parallelism. Since graphs can be highly irregular, however, such reductions in parallelism are not always desirable. Cong et al therefore propose a heuristic for controlling more carefully the number of vertices in a batch. The basic idea of the heuristic is to create full batches when the processors are busy (there is much work) and create small, partially-filled batches when the processors are idling (there is little work). It is not known whether the heuristic can provably control the overheads without overly limiting parallelism. Indeed, it might not; for example, our experiments show that the heuristic can lead to suboptimal performance (Section 6).

In this paper, we present a provably work-efficient and highly parallel algorithm for performing PDFS. The three key techniques behind the algorithm are a novel data structure called *splittable weighted frontier data structure* for representing the frontiers, a novel amortization technique for controlling granularity of parallelism, and an adaption lazy splitting to the unordered parallel DFS problem.

Splittable Weighted Frontier. The splittable weighted frontier data structure allows operating on the frontiers by using, for example, push and pull operations that act on vertices and edges. These operations are efficient both in theory and in practice (Section 3).

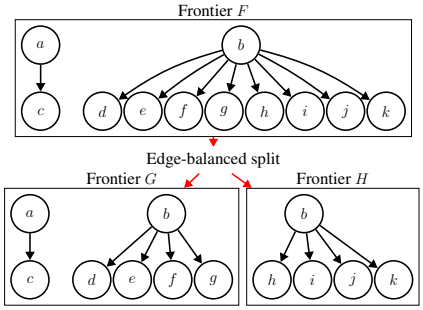

Figure 1: The edge-balanced split operation on the frontier. The frontier F consists of the vertices a and b and implicitly their out-edges, c through k. Performing a split operation divides the frontier into two frontiers, with equal number of outgoing edges (within a margin of 1), by dividing the edges of the vertices as necessary.

By taking advantage of the splittable weighted frontier data structure, our PDFS algorithm incurs relatively small overheads compared to a serial DFS algorithm that uses a simple stack data structure to represent the frontier. Furthermore, the splittable weighted frontier data structure supports an efficient *balanced-split* operation that splits the frontier into two halves based on the weights—the outdegrees—of the frontier vertices. Our PDFS algorithm distributes work using the balanced split operation: when an idle processor steals from another target processor, the target processor splits its frontier and sends one of the resulting halves to the idle processor. Such split operations allow sharing work at the granularity that is essentially optimal based on local information.

The design and the implementation of the splittable weighted frontier builds on a *chunked sequence* data structure we introduced in prior work [7]. One key idea behind the data structure is to use a hierarchical representation that allows operating at the level of vertices as well as edges. For example, a vertex can be inserted into the frontier along with all of its edges and a frontier can be split into two halves based on the total number of edges (the degree of the vertices in the frontier). Figure 1 illustrates an example edge-balanced split operation. Each resulting half contains a single non-empty carry consisting of a half of the edges of vertex b. Prior work showed that it can be important to control the degree of vertices and proposed vertex-virtualization techniques [43]. The splittable weighted frontier data structure can be viewed as performing on-demand vertex-virtualization; it thus combines the best of vertex-based and edge-based parallelization techniques.

Granularity control. In PDFS, as in all parallel algorithms, there is an inherent tradeoff between two quantities: (1) the cost of migrating a piece of work from one processor to another; and (2) the benefit of parallelism that can be gained by such work migration. In divide-and-conquer and similar parallel algorithms, this tradeoff can be solved by a technique that parallelizes or sequentializes tasks based on well-informed estimates of how long pending tasks may take to complete. In this regime, a task may be migrated only if the granularity-control algorithm chooses to parallelize the task [5]. In PDFS, this approach does not apply because there is no efficient way to estimate (a priori) the amount of work, specifically the work that would be performed by visiting a vertex.

To see this problem more concretely, we will use two simple example graphs; it is not difficult to generalize the examples to more interesting examples that exhibit essentially the same problem at a larger scale. Consider the graph shown in Figure 2. Two processors can traverse the two long chains in the graph in parallel, leading to

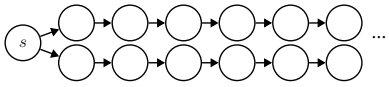

Figure 2: Example for aggressive work sharing.

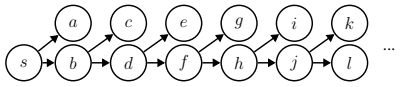

Figure 3: Example against aggressive work sharing.

a 2x speedup. To take advantage of the parallelism offered by this input graph, the algorithm must split a frontier consisting of two edges and migrate one of the vertices to another processor. The trouble is that splitting such small frontiers can lead to significant overheads in other graphs. To see why, consider as an example the graph shown in Figure 3. Assume processor 1 begins execution with the source s, sends vertex b to processor 2, and continues working on vertex a. Soon enough processor 1 runs out of work, because a has no outgoing edges. By this time, processor 2 has processed b and has a frontier that consisting of c and d, so it sends d to processor 1. Soon enough, processor 2 runs out of work and we are back to the same situation as in the beginning, but this time with d as the new source. This second example shows that systematically splitting small frontiers can lead to a large number of communication operations. In summary, splitting small frontiers is essential for parallelism in some graphs, but leads to high overheads in others.

To solve this granularity-control problem in PDFS, we propose a technique based on amortization: each processor shares work only if either (1) it has a large frontier, or (2) the processor has already performed some predetermined amount of work locally since the last time it shared work. This technique enables our PDFS algorithm to share even as little as a single edge, while amortizing the cost of task migration against either the amount of work that will be performed in the future (by traversing the vertices in the frontier, i.e., case 1) or the amount of work that has been performed in the past locally by the processor (case 2). Our algorithm is thus able to achieve high parallelism and work efficiency at the same time.

Lazy Splitting. By using the splittable weighted frontier data structure and the granularity-control techniques proposed here, it is possible to present a PDFS algorithm that generates parallelism by eagerly splitting the frontiers and sharing the resulting pieces with other processors via work stealing. Such an eager approach suffers from overheads that come from splitting and exposing pieces of parallel work that are not necessarily ever shared. In order to reduce overheads further and to reach tight running-time bounds, we instead solve this problem by using lazy spitting [48]. In lazy splitting, instead of creating parallelism eagerly, processors create parallelism only if there is a demand for parallelism. Specifically, our PDFS algorithm splits a frontier only if another processor requests work from it.

Analysis. Based on these techniques, we establish two theorems. The first theorem establishes a *responsiveness* property by showing that, when it is queried for work, a processor responds quickly, either by rejecting the query (when it has no work to share, or when it refuses to share work in order to control communication overheads), or by sharing half of the work load in its frontier. This property establishes that the algorithm generates parallelism quickly and as needed. Our second theorem establishes *strong work efficiency*: we prove that, when we explicitly account for the scheduling overheads (including the cost of polling for queries, the cost of answering to queries, and the cost of splitting frontiers), the asymptotic complexity of PDFS remains $O(n + m)$, i.e., same as that of sequential DFS.

3 Splittable Weighted Frontier

We present the implementation of our splittable weighted frontier data structure. Our PDFS algorithm uses the splittable weighted frontier data structure to represent the most recently visited vertices in depth-first traversal. The data structure supports the following operations:

- `empty` returns a boolean indicating whether the data structure is empty;
- `nb_edges` returns the cardinality of the frontier;
- `push_edges_of` pushes all the out-edges of the given vertex into the frontier;
- `iter_pop_nb` iterates a given operation over `nb` edges (or fewer, depending on the availability) and removes from the frontier each edge considered;
- `split` carves out half of the edges into an independent frontier data structure.

To support these operations efficiently, we use a recently proposed weighted-sequence data structure. We describe this data structure next.

Splittable weighted sequences. A splittable weighted-sequence data structure supports pushing items into and popping items from the two ends of the sequence, allows assigning a weight to each item, and splitting sequences at a specified weight.

Recent work [7] gives an asymptotically efficient and practically fast splittable weighted-sequence data structure by using a chunking and a bootstrapping technique that allows representing the sequence data structure as a shallow tree. The data structure, called *bootstrapped chunked sequence*, stores a sequence of weighted items. Perhaps the most interesting operation for our purposes is the operation `split_at`, which takes a weight w and a sequence S and partitions S into three parts: S_1, $\{x\}$, and S_2, in such a way that the total weight of S_1 is less than w and that the weight of $S_1 \cup \{x\}$ is greater than or equal to w.

Bootstrapped chunked sequences ensure practical efficiency by storing items in fixed-capacity chunks (represented as arrays). A *chunk size* parameter, called B, controls the size of the chunks. For a given $B \geq 2$, the cost of split operations is bounded by $O(B \cdot \log_B n)$. Besides, the worst-case asymptotic space usage of chunked sequences is $(1 + \frac{O(1)}{B}) * n$, which, for any sufficiently-large value of B, is close to optimal.

Implementation of splittable edge-weighted frontiers. We implement splittable edge-weighted frontiers on top of bootstrapped chunked sequence. The basic idea is to represent a frontier as a triple consisting of *vertex-sequence* and two *ranges* of edges. A vertex sequence is represented as a bootstrapped chunked sequence of vertices, where each vertex has a weight that matches its out-degree. A range of edges corresponds to a contiguous subset (subsequence) of the outgoing edges of a given vertex. A range is represented as a vertex and a pair of indices marking the start and the stop of the range.

Figure 4 shows the implementation of the range data structure, which admits a straightforward implementation, and that of the frontier data structure, which we describe next. In the following discussion, we treat the adjacency list, called `neighbors` in the pseudocode, as a global variable; in the actual implementation the definitions are parameterized by the adjacency list structure.

```
bag<int> neighbors[nb_vertices] // adjacency lists

class range
  int vertex; int low; int hi;

  range() { vertex = 0; low = 0; hi = 0 }

  weight() { return hi-low }

  void split_at(int w, range& other)
    other.vertex = vertex
    other.low = low + w
    other.hi = hi
    hi = low + w

  int iter_pop_nb(int nb, body_type body)
      where body_type = void body(int src, int dst)
    if nb == 0 then return 0
    nb = min(nb, hi-low)
    int stop = low + nb
    for k = low to stop-1
      body(vertex, neighbors[vertex][k])
    low = stop
    return nb

class frontier
  weighted_seq<int> vs; range r1; range r2;

  frontier()
    vs = weighted_seq<int>(fun v → degree(v))
    r1 = range()
    r2 = range()

  int degree(int vertex)
     return neighbors[vertex].size()

  range full_range(int vertex)
     return range(vertex, 0, degree(vertex))

  int nb_edges()
    return vs.weight() + r1.weight() + r2.weight()

  bool empty()
    return nb_edges() == 0

  void push_edges_of(int vertex)
    if degree(vertex) > 0
      vs.push(vertex)

  void split(frontier& other)
    int w = (nb_edges()+1) / 2
    if w <= r1.weight()
      r1.split_at(w, other.r1)
    else if w <= r2.weight()
      r2.split_at(w, other.r1)
    else
      w -= r1.weight()
      other.r2 = r2
      int v;
      vs.split_at(w, v, other.vs)
      r2 = full_range(v)
      r2.split_at(w - vs.nb_edges(), other.r1)

  int iter_pop_nb(int nb, body_type body)
    nb -= r1.iter_pop_nb(nb, body)
    while nb > 0 && not vs.empty()
      int vertex = vs.pop()
      int deg = degree(vertex)
      if deg <= nb
        range r = full_range(vertex)
        nb -= r.iter_pop_nb(nb, body)
      else
        r1 = full_range(vertex)
        nb -= r1.iter_pop_nb(nb, body)
        return nb
    nb -= r2.iter_pop_nb(nb, body)
    return nb
```

Figure 4: Implementation of the frontier data structure.

```
const int D // controls the frequency of polling
const int K // controls the eagerness of work sharing

// global shared array for marking visited vertices
bool visited[nb_vertices] = { false, false, ... }

void parallel_dfs_thread()
  frontier fr = frontier()
  int nb = 0
  while true do
    if fr.empty()
      if traversal_completed()
        return
      nb = 0
      acquire(&fr)
    else
      if has_incoming_query()
        int sz = fr.nb_edges()
        if (sz > K) || (nb > K && sz > 1)
          reply(fun (frontier* other_fr) →
                  fr.split(*other_fr))
          nb = 0
        else
          reject_query()
      nb +=
       fr.iter_pop_nb(D, fun(v,target)→
         if (not visited[target]
             && cas(&visited[target], false, true))
           fr.push_edges_of(target))
```

Figure 5: PDFS code executed by each processor.

The operation `frontier` constructs an empty frontier. The operation `nb_edges` returns the number of edges in the frontier, computed as the total weight of the vertex-sequence plus the sum of the width of the two ranges. The operation `empty` returns whether the number of edges in the frontier is nonzero. The operation `push_edges_of` pushes the vertex given to the vertex-sequence if it has outgoing edges associated with it.

The operation `split` transfers half—the smaller half in case the cardinality is not even— of the edges to another frontier data structure, which is assumed to be initially empty; it leaves the other half in place. The operation is implemented as follows. If the first range contains at least half of the edges, we simply split this range and transfer a subrange to the other frontier. Else, if the second range contains at least half of the edges, we tranfer the appropriate subrange from it. Otherwise, we need to split the sequence of vertices. First, we transfer all of the second range to the other frontier. Then, we split the sequence of vertices in three parts: vertices that remain in the vertex-sequence, vertices that go into the vertex-sequence of the other frontier, and one vertex which contains the median edge. We consider the full range of edges associated with this vertex and split this range at the appropriate position, storing the left subrange into the second range of the current frontier and storing the right subrange into the first range of the other frontier.

The function `iter_pop_nb` iterates over at most `nb` edges, popping them from the frontier as it processes them. It returns the number of edges effectively processed. The edges considered are first picked from the first range, then from the edges associated with the vertices stored in the vertex sequence, and finally from the second range. Note that if a vertex has a large arity, it is possible that only a fraction of its edges are processed; in such case, the remaining edges are placed into the first range, which must be empty in this case. The challenge in implementing this function is that efficiency is critical in the loop over the edges—we are careful to limit the number of operations performed compared with the corresponding loop in the sequential DFS algorithm.

4 Parallel Depth-First Search

Our PDFS algorithm uses the splittable weighted frontier data structure to represent the most recently visited vertices in depth-first-search or the frontier. The basic idea behind our algorithm is to distribute the most recently visited vertices, i.e., the frontier, across the processors, each of which holds its portion of the frontier in a local splittable weighted frontier data structure and explores the graph in a depth-first manner starting from the frontier. In order to perform effective load balancing and granularity control, each processor alternates between two phases: *working* and a *load-balancing*. In the working phase, the processor removes some number of vertices from its frontier, visits them, and adds their neighbors to its frontier. The number of vertices removed is determined by the polling parameter, written as D. In the load-balancing phase, the processor performs either one of two actions based on whether the frontier is empty or not.

- **Case 1: the frontier is empty.** If the traversal is not complete, the processor requests work from another processor by sending that processor a work-request message.
- **Case 2: the frontier is nonempty.** If it has an incoming work-request message, the processor responds to the request by either sending work or rejecting the request.

The processor responds positively to a work request from another processor only if the processor has done sufficiently large amount of work or if the processor has sufficiently many vertices in its frontier to make sharing worthwhile. The "sufficiency" condition is guided by a granularity parameter, written as K.

Crucial to the effectiveness of the algorithm is the choice of the polling and granularity parameters, D and K respectively. To ensure effective load balancing, the polling parameter D should be just large enough to amortize the cost of polling for queries. The granularity parameter K should be just large enough to amortize the cost of splitting and communicating work.

Figure 5 shows the pseudo-code for the algorithm being executed by each of the processors taking part in a run of our PDFS. We assume the graph to be represented by an adjacency list and use an array of booleans, which we call `visited`, to mark the vertices that have been visited. Each processor maintains the portion of the frontier that it is working on and keeps track in a variable named `nb` of the number of edges the processor has processed since the previous load balancing operation. When the local frontier is empty, termination is tested by calling the function `traversal_completed`. Until the traversal is complete, each processor is busy performing one of three actions: (1) it is working on its own frontier, or (2) it is requesting work from another processor by calling a function named `acquire` in order to make queries to busy processors, or (3) it is responding to a work query.

A processor with an empty frontier calls the blocking function `acquire`, passing it the address of its frontier so that target processor may directly transfer data into the frontier. In the load balancing scheme that we consider, the `acquire` function targets a single processor at a time, blocks until an answer is received, and repeats until obtaining work. While acquiring work, the processor rejects any incoming query from other processors —this behavior can be implemented, e.g. by writing a dummy value into the query cell, so as to prevent any query to be made.

To work on its frontier, the processor visits the edges in its frontier and adds the outgoing edges of each visited vertex to its frontier. To test whether a vertex is visited, the processor first executes a conventional read. If the vertex appears to be previsouly unvisited, it performs an atomic compare-and-swap (CAS) operation to mark the vertex visited.

In order to perform load balancing actions, each processor calls the function `has_incoming_query` after visiting D edges. If it finds an incoming query from an idle processor, the processor needs to reply to the query either by rejecting it, using the function `reject_query`, or by transfering work, using the function `reply`. The latter is presented using a callback argument, which allows the processor to obtain the address of the empty frontier data structure where it should migrate edges.

A processor that finds an incoming query from an idle processor accepts to share its frontier only if either of the following conditions hold: (1) its frontier contains more than K edges, or (2) it has locally processed more than K edges since the last work transfer (and it has at least one edge to send). The first condition corresponds to the classical granularity-based approaches to amortizing cost of thread creation by charging to the amount of work that *will be* performed by the DFS algorithm on that frontier. As we described earlier, however, the first condition alone does not successfully expose the parallelism available in the graph. The second condition solves this problem by amortizing the cost of thread creation to the work that *has already been performed* locally by checking that it has processed at least K many edges. This bi-directional (future and past) amortization technique thus allows us to create threads for work that may be tiny, yet still amortize the cost of thread creation.

The PDFS traversal terminates when the frontiers of all processors become empty. The termination-detection problem is essentially orthogonal to our discussion, so we only describe it briefly. A naive approach is to rely on a global atomic counter, keeping track of the number of processors with a nonempty frontier (for details, see [20], Section 2.2). While this approach may work well on small machines with a few dozen processors, we expect that scaling up to a larger number of processors would require a more advanced termination detection strategy, for example one based on hypercube or lifeline network graphs [42], in order to distribute among several processors the effort of checking whether all processors have run out of work. In our implementation, we use a refinement of the naive approach.

5 Analysis

We next present an analysis of our PDFS algorithm that takes into account important scheduling costs, such as the time to create parallel threads (splitting frontiers and forking jobs) and the time to communicate (polling on queries). We prove that our algorithm, while also ensuring good load balance by splitting the work of a busy processor equally when demanded by another, is able to achieve work efficiency by amortizing those potentially large costs.

We consider the usual RAM model, where each instruction takes constant time to execute, except for scheduling-related instructions, for which we introduce two specific parameters, called C_{fork} and C_{poll}, as described next.

Definition 5.1 *Our analysis uses the following parameters:*

- *n and m denote the number of vertices and edges in the graph.*
- *P denotes the number of processors (cores).*
- *C_{fork} is an upper bound on the cost of transfering a frontier (but excluding the cost of splitting the frontier).*
- *C_{poll} is an uppert bound on the cost of polling and responding to a query (but excluding the cost of splitting and transfering the frontier).*
- *B denotes the size of a chunk in vertex-sequences ($B \geq 2$).*
- *D is a positive integer controlling the frequency of polling.*

- *K is a positive integer controlling the eagerness of work sharing.*

Below, we call size of a frontier (and write f) for the number of edges stored in the frontier considered.

We begin with results on the frontier data structure. Based on the known bounds of the vertex-sequence data structure, and since basic operations on ranges are constant time, it is straightforward to prove the following theorem.

Theorem 5.1 (Efficiency of the frontier data structure) *Our frontier data structure admits the following bounds:*

- *The allocation of an empty frontier is $O(B)$.*
- *`nb_edges` is $O(1)$.*
- *`push_edges_of` is $O(1)$.*
- *`split` is $O(B \log_B v)$, where v is the number of vertices stored.*
- *`split` is also $O(B \log_B f)$, where f is the size of the frontier, i.e. the number of edges it contains, because the frontier only stores vertices with positive outdegree.*
- *`iter_pop_nb` costs $O(1)$ per edge enumerated (plus the cost of the function processing the edges).*
- *The asymptotic space usage is $(1 + \frac{O(1)}{B}) * f$, close to optimal.*

For PDFS, we first establish a responsiveness property of our load balancing scheme. This property ensures that parallelism is generated quickly, when needed (as requested by idle processors).

Theorem 5.2 (Responsiveness) *If a processor receives a query, then it either rejects it within a delay $D + O(1)$, or it responds by sharing work within a delay $D + O(B \log_B n)$.*

PROOF. If the processor receiving the query is running the function `acquire`, then it rejects the query in $O(1)$. If it is already serving a query (possibly performing a split operation), then its query cell is occupied and it cannot receive a new query. If a processor is working on its frontier, then it processes at most D edges before it polls. When a processor polls and finds an incoming query, it either rejects it, after a delay $O(1)$; or it splits its frontier, and does so in time $O(B * \log_B v)$ with $v \leq n$, according to Theorem 5.1. □

We next establish two key lemmas to bound the number of split operations and to bound the total cost of the split operations involved in a PDFS execution.

Lemma 5.3 (Maximal number of frontier split operations) *Our PDFS algorithm performs at most $\frac{3m}{K}$ split operations.*

PROOF. For this proof, we introduce a per-processor potential function satisfying the following properties: (1) the potential is always nonnegative; (2) when inserting an edge into the frontier, the potential increases by at most $\frac{3}{K}$ units; (3) when removing an edge from the frontier, the potential does not increase; (4) when splitting a frontier to share work with another processor, the total potential decreases by at least one unit. Overall, since at most m edges can be inserted into the frontiers in any PDFS execution, the total increase in potential is at most $\frac{3m}{K}$. Since every split operation decreases the total potential by at least one unit, there can be at most $\frac{3m}{K}$ splits.

Consider a processor having a frontier storing f edges and a local variable `nb`. We define its potential, written $\phi(f, \mathtt{nb})$, as the value $\frac{1}{K}(f + \mathtt{nb} + 2 \cdot (f - \frac{K}{2})^+)$, where $(x)^+$ denotes $\max(0, x)$. We next prove the desired properties.

(1) The potential is always nonnegative, because $f \geq 0$ and $\mathtt{nb} \geq 0$. (2) When inserting an edge into the frontier, f increases by one; the increase in potential is $\phi(f + 1, \mathtt{nb}) - \phi(f, \mathtt{nb}) \leq \frac{3}{K}$. (3) When removing an edge from the frontier, `nb` increases by one and f decreases by one; the potential does not increase because $\phi(f - 1, \mathtt{nb} + 1) - \phi(f, \mathtt{nb}) \leq 0$. (4) It remains to consider the case of a split, for which we wish to prove that the total potential decreases by at least one unit. Considering the potential of the sender and that of the receiver, the goal is to prove: $\phi(f, \mathtt{nb}) + \phi(0, 0) \geq 1 + \phi(\lceil \frac{f}{2} \rceil, 0) + \phi(\lfloor \frac{f}{2} \rfloor, 0)$. This inequality is equivalent to: $\frac{\mathtt{nb}}{K} + \frac{2}{K}\left((f - \frac{K}{2})^+ - (\lceil \frac{f}{2} \rceil - \frac{K}{2})^+ - (\lfloor \frac{f}{2} \rfloor - \frac{K}{2})^+\right) \geq 1$. There are two conditions under which a split operation may be triggered: $f > K$ or $\mathtt{nb} > K$. We consider each case separately.

First, assume $f > K$. In this case, we have $f \geq \lceil \frac{f}{2} \rceil \geq \lfloor \frac{f}{2} \rfloor \geq \frac{K}{2}$. The desired inequality is equivalent to: $\frac{\mathtt{nb}}{K} + \frac{2}{K} \cdot (f - \lceil \frac{f}{2} \rceil - \lfloor \frac{f}{2} \rfloor + \frac{K}{2}) \geq 1$, which is true since $f = \lceil \frac{f}{2} \rceil + \lfloor \frac{f}{2} \rfloor$ and $\frac{\mathtt{nb}}{K} \geq 0$.

Second, assume $\mathtt{nb} > K$. We have $\frac{\mathtt{nb}}{K} \geq 1$. Thus, to establish the desired inequality, it suffices to show: $(f - \frac{K}{2})^+ - (\lceil \frac{f}{2} \rceil - \frac{K}{2})^+ - (\lfloor \frac{f}{2} \rfloor - \frac{K}{2})^+ \geq 0$. To see why this inequality holds, we distinguish two cases. If $\lfloor \frac{f}{2} \rfloor \geq \frac{K}{2}$, then the inequality simplifies to $\frac{K}{2} \geq 0$. Otherwise, we have $\lfloor \frac{f}{2} \rfloor < \frac{K}{2}$, in which case $(\lfloor \frac{f}{2} \rfloor - \frac{K}{2})^+ = 0$, and we are able to conclude by observing that: $(f - \frac{K}{2})^+ \geq (\lceil \frac{f}{2} \rceil - \frac{K}{2})^+$.

In summary, both conditions that may trigger a split operations ensure that the total potential decreases by at least one unit. □

Lemma 5.4 (Maximal cost of the split operations) *The total cost of all split operations in a PDFS execution is $O(\frac{mB \log_B(4K)}{K})$.*

PROOF. Due to space limitation, the details of the proof are not included here; they may be found in the technical appendix[1]. The proof follows a potential analysis similar to that of the previous lemma. The potential function, written $\Phi(\mathtt{nb}, f)$, is defined as:

$$\frac{rB(\mathtt{nb}+f)}{K} \log_B K \; + \; \mathsf{if}\; f \leq \tfrac{K}{2}\; \mathsf{then}\; 0\; \mathsf{else}\; rB(\tfrac{a}{K} f - \log_B f - b)$$

where $a = 2 \log_B K + 6 \log_B 2$ and $b = 4 \log_B 2$, and where r is such that the cost of the split of a frontier of size f is bounded by $rB \log_B f$. We prove, in particular, that each edge insertion increments the potential by no more than $\frac{3rB}{K} \log_B(4K)$ units. □

We are now ready to establish our work-efficiency theorem. The theorem shows that our algorithm is asymptotically work efficient with respect to serial DFS and that the potentially expensive costs (in particular, C_{fork} and $B \log_B 4K$) are amortized by the parameters K and D, which can be set to sufficiently-large values to ensure limited load balancing overheads.

Theorem 5.5 (Work efficiency) *The total work performed by PDFS is bounded by:*

$$O\left(n + \left(1 + \frac{C_{\mathsf{fork}}}{K} + \frac{C_{\mathsf{poll}}}{\mathsf{min}(D, K)} + \frac{B \log_B(4K)}{K}\right) m + BP\right).$$

When all parameters are fixed, the total work is $O(n + m)$.

PROOF. Consider the pseudo-code in Figure 5. The operations that contribute to the total work are: (0) the allocation of the frontier data structures; (1) the calls to `acquire`; (2) the polling operation and possibly its subsequent rejection of the incoming query; (3) the splitting of the frontier; (4) the delivery of the splitting frontier;

[1] Proof appendix available from: http://deepsea.inria.fr/pdfs-sc15 .

(5) the popping of edges from the frontier and read of the status of the target vertex; (6) the addition of new vertices into the frontier. To analyse the cost of each of these contributions, keep in mind the result of Lemma 5.3, which bounds the number of split operations by $\frac{3m}{K}$. In particular, the total number of calls to `acquire` is bounded by $\frac{3m}{K} + O(P)$, because there is one initial call per processor, and then each call to acquire many only return as a result of a split operation.

For contribution (0), the initialization of each frontier costs $O(B)$, so the total cost is $O(BP)$. For (1), since each call to `acquire` costs $O(1)$ work, and since there are at most $\frac{3m}{K} + P$ such calls, the total work associated with `acquire` is thus $O(\frac{3m}{K} + P)$. For (2), observe that a polling operation, of cost C_{poll}, takes place either after D edges have been processed, or immediately after receiving a frontier. Thus, the total work induced is $O(\frac{m}{D}C_{\mathsf{poll}} + \frac{3m}{K}C_{\mathsf{poll}})$, which is, $O(m\frac{C_{\mathsf{poll}}}{\min(D,K)})$. For (3), we exploit Lemma 5.4, which shows that the total cost of split operations is $O(\frac{mB\log_B(4K)}{K})$. For (4), we multiply the cost C_{fork} with the maximal number of splitting operations; thus, the total cost of forks is $O(\frac{3m}{K}C_{\mathsf{fork}})$. For (5), extracting each edge from the frontier costs $O(1)$ by Theorem 5.1, and checking the target vertex of each edge also costs $O(1)$, so the total cost is $O(m)$. For (6), adding each vertex to the frontier costs $O(1)$, also by Theorem 5.1, so the total cost is $O(n)$. Summing up all costs involved gives the bound stated in the theorem. □

6 Implementation and Experiments

6.1 Implementation and Experimental Setup

We implemented our benchmarking program in C++, using Pthreads to realize parallelism. At runtime, our benchmarking program first loads the input graph and then spawns one Pthread for each processor in the machine. For the implementation of our PDFS algorithm, we start each of the Pthreads running an instance of the `parallel_dfs_thread` function (shown in Figure 5). To begin the graph traversal, we populate the frontier of one arbitrary processor with the source vertex. We implemented the load balancing functions `has_incoming_query`, `reply`, `acquire`, and `reject_query` using a simple protocol that bears close resemblance to the one that is used by the private-deques work-stealing algorithm [6]. In the protocol, work-request and acknowledgement messages are sent and received via the atomic-cell structures provided by the `std::atomic` library of C++11.

Recall that our spilttable weighted frontier data structure depends on an underlying bootstrapped chunked sequence data structure. For this underlying structure, we used the same C++ implentation that we used in the experimental evaluation of our own bootstrapped chunked sequence [7].

The implementation represents graphs using the "compressed adjacency list", in which vertices are labeled with natural numbers in the range $[0, \ldots, n-1]$, where n is the number of vertices in the input graph. In this representation a graph is represented as a single array of 32-bit or 64-bit cells (depending on the size of graph). The array starts with a sequence of offsets, with one offset per vertex, followed by a sequence of vertex ids. The offset entry in the first sequence for vertex v marks the starting position in the second sequence for the list of outgoing edges of v.

For the experiments, we use a few parameters that are specific to the target system architecture (but not specific to the input). The constant D is used to amortize the cost of a single read and therefore essentially any moderately large constant (in the hundreds) reduces the overhead to less that 1%. In our implementation, D is set to 256. The constant K is used to amortize the cost of coordination with the scheduler and communication between threads, which involves several reads and writes. Therefore a constant slightly larger than D suffices to reduce the overheads approximately to less than 1%. In our implementation K is set to 1024. In chunked sequences, we use chunks of size $B = 32$, except for the first layer of the data structure which, as an optimization, uses chunks of size 1024. With these parameters, the cost of a split on a frontier of size f is of the form $O(1024 + 32\log_{32} f)$. The constant factors involved are relatively small thanks to the use of highly-optimized *memcpy* operations for manipulating the chunks.

Our experience with modern multicore machines, which are equipped with non-uniform memory architecture (NUMA), show that allocation policy can have a significant impact on performance. We therefore control allocation by determining the memory bank (in NUMA machines, there is typically one memory bank per chip) at which objects are allocated. For sequential programs, we allocate all pages on the memory bank closest to the chip, which runs the program. This policy gives the best performance by placing the objects at the closest possible location in memory. For parallel programs, we allocate pages across memory banks in a round-robin fashion, thereby balancing memory traffic across chips; this policy appears to give the best performance for parallel runs.

We compiled all programs with GCC version 4.9.1, using optimizations `-O2 -march=native`. For the measurements, we considered an Ubuntu Linux machine with kernel v3.2.0-58-generic. For scalable heap allocation, we used `tcmalloc` from gperftools version 2.4. Our benchmark machine has four Intel E7-4870 chips and 1Tb of RAM. Each chip has ten cores and shares a 30Mb L3 cache. The main main memory of the machine is distributed across four banks: one per chip. Each core runs at 2.4Ghz and has 256Kb of L2 cache and 32Kb of L1 cache. Additionally, each core hosts two SMT threads, giving a total of eighty hardware threads. However, to avoid complications with hyperthreading, we did not use more than forty threads.

In order to reduce the impact of noise on our results, we average our measurements over 10 runs. In a few cases, the noise in a single run was as high as 10%, but in other cases, noise was below 5%. Sequential runs showed negligible variance.

6.2 Input Graphs

Table 1 summarizes graph inputs.

We considered the following large publicly available graphs that come from data that was sampled from the real world. The *orkut*, *livejournal*, *twitter* and *friendster* graphs describe social networks [31, 1]. The *wikipedia* (as of 6 February 2007), *Freescale1*, and *cage15* graphs are taken from the University of Florida sparse-matrix collection [21]. The *rgg* ($n = 23$), *delaunay* ($n = 24$), *usa* (full), and *europe* (full) graphs are taken from DIMACS challenge problems [2, 3].

We also consider a set of synthetic graphs that we selected to range from moderately to highly parallelizeable. For these graphs, we assign each vertex a unique, randomly chosen number in the range $[0, \ldots, n-1]$. This random assignment prevents accidental effects of alignment from our measurements.

The *square-grid* and *cube-grid* graphs are directed grids in two- and three-dimensional space in which each vertex has 2 and 3 outgoing edges, respectively. The *random-arity-100* graph is a uniform random graph, with average arity 100 on every vertex. The *complete-bin-tree* graph is a perfect binary tree. The *rmat24* and *rmat27* graphs are synthetic graphs with power-law distribution degrees [16]. The former was generated using settings $a = 0.5$, $b = c = 0.1$, and $d = 0.3$, and the latter with $a = 0.57$, $b = c = 0.19$ and $d = 0.05$.

graph	vertices (m)	edges (m)	vertices seen	edges seen	max dist.	seq.DFS (s)	PDFS 1-core	PDFS (s)	PDFS vs seq.	seq.DFS (mEdge/s)	PDFS (mEdge/s)
orkut	3.1	117	>99%	100%	7	0.97s	+34%	0.06s	15.7x	121	1902
livejournal	4.8	69	91%	99%	14	1.13s	+49%	0.06s	18.2x	60	1093
twitter	42	1468	84%	96%	15	24.29s	+68%	1.19s	20.4x	58	1191
friendster	125	1806	52%	>99%	28	55.91s	+29%	2.16s	25.8x	32	831
cage15	5.2	99	100%	100%	49	1.25s	+42%	0.07s	17.4x	79	1378
Freescale1	3.4	19	99%	>99%	122	0.24s	+73%	0.03s	8.4x	77	642
wikipedia-2007	3.6	45	67%	93%	459	0.73s	+42%	0.05s	15.6x	58	902
rgg	8.4	127	>99%	>99%	1.5k	1.24s	+42%	0.16s	7.7x	103	795
delaunay	17	101	100%	100%	1.6k	1.30s	+50%	0.09s	15.0x	77	1163
usa	24	58	>99%	100%	6.3k	1.31s	+52%	0.09s	14.5x	44	646
europe	51	108	100%	100%	17k	2.62s	+50%	0.16s	16.1x	41	667
trees-10k-10k	100	100	100%	100%	2	7.08s	+60%	0.43s	16.6x	14	234
random-arity-100	1.0	100	100%	100%	4	0.89s	+29%	0.05s	19.2x	112	2158
rmat27	17	119	34%	98%	6	3.31s	+45%	0.15s	21.7x	35	767
phases-10-d-2	33	93	100%	100%	10	13.24s	+6%	0.41s	32.6x	7.0	230
rmat24	17	120	90%	98%	13	6.44s	+56%	0.29s	22.1x	18	403
phases-20-d-100	5.0	475	100%	100%	20	6.42s	+26%	0.23s	28.2x	74	2089
complete-bin-tree	134	134	>99%	100%	26	33.10s	+12%	2.12s	15.6x	4.1	63
cube-grid	33	99	100%	100%	960	10.37s	+50%	0.44s	23.7x	9.6	226
trees-524k	200	200	100%	100%	381	15.49s	+49%	0.89s	17.4x	13	224
square-grid	50	100	100%	100%	14k	13.36s	+53%	0.58s	23.0x	7.5	172
par-chains-100	50	50	100%	100%	500k	17.95s	+29%	0.73s	24.6x	2.8	69
trunk-first	10	10	>99%	100%	10.0m	3.15s	+30%	4.13s	0.8x	3.2	2.4
par-chains-2	50	50	100%	100%	25m	17.62s	+28%	11.35s	1.6x	2.8	4.4
chain	50	50	100%	100%	50m	17.45s	+29%	22.79s	0.8x	2.9	2.2

Table 1: Input graphs description, raw execution times for the baseline algorithms and for our parallel algorithms, and throughput.

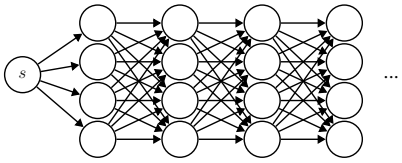

Figure 7: Example synthetic graph: phases-4-d-4.

The *chain* graph is a single, long path. The *par-chains-x* graphs are different instantiations of the pattern shown in Figure 2, where x denotes the number of independent maximal chains originating from the root. The *trunk-first* graph is an instance of the challenge graph shown in Figure 3. The graph *trees-524k* is a generalization of the former: it consists of a main chain of length 381, where each vertex has 524,288 outedges.

The *trees-10k-10k* graph is a tree with two levels, in which the root node is connected to 10k children and each of these child nodes connects to 10k leaf nodes. This graph tests the ability of the algorithms to exploit parallelism in the lists of neighbors of the vertices. The *phases-x-d-y* graphs are instances of the structure shown in Figure 7. These graphs generalize the idea of the grids, thus allowing us to have an even smaller number of frontiers (e.g., 10, or 20), and control the arity of the vertices (e.g., 2, or 100). In the graph, *phases-10-d-2*, each of the 10 frontiers contains 3.3 million vertices and each vertex has arity 2, except one particular vertex in each frontier, which is linked to all the vertices in the next frontier (and thus has arity 3.3 million). The goal of these graphs is to stress the need for splitting the frontier according to the number of edges and not just the number of vertices.

6.3 Comparison with Baseline Sequential DFS

We first compare our algorithm to an implementation of the sequential DFS algorithm, which serves as the baseline throughout the paper. We carefully optimized our implementation of the sequential DFS, for example, by using a fixed-capacity array for storing vertices. Using a fixed-capacity array enables excluding overheads associated with array-resize operations; realistic implementations of DFS often rely on resizable stacks in order to limit the space usage.

The left half of the graph reports, for each graph traversal from the source vertex, the number of reachable vertices and edges, and the maximal distance from the source over all reachable vertices. The right half of the table reports results on sequential DFS and our PDFS.

An important property of our algorithm, as proved in Section 5, is that it controls scheduling overheads by using a novel frontier data structure and by using amortization techniques. To validate the theory, we measured our PDFS algorithm on a single core, which allows measuring some part of the total work that our parallel algorithm performs during parallel runs. This measure does not include the cost of splitting and migrating frontiers, but includes the cost of using the frontier data structure, the cost of regularly polling on queries, and the cost due to the NUMA allocation policy employed in non-sequential runs. The single-core run thus gives us a lower bound to the work overheads of our PDFS algorithm. The data reported in the column labeled "PDFS 1-core" in Table 1 shows that the overheads compared with the baseline are 40% on average. We separately verified that half of this overhead is due to the NUMA round-robin allocation policy. Thus, the overhead due to the use of the frontier data structure and polling is approximately 20% on

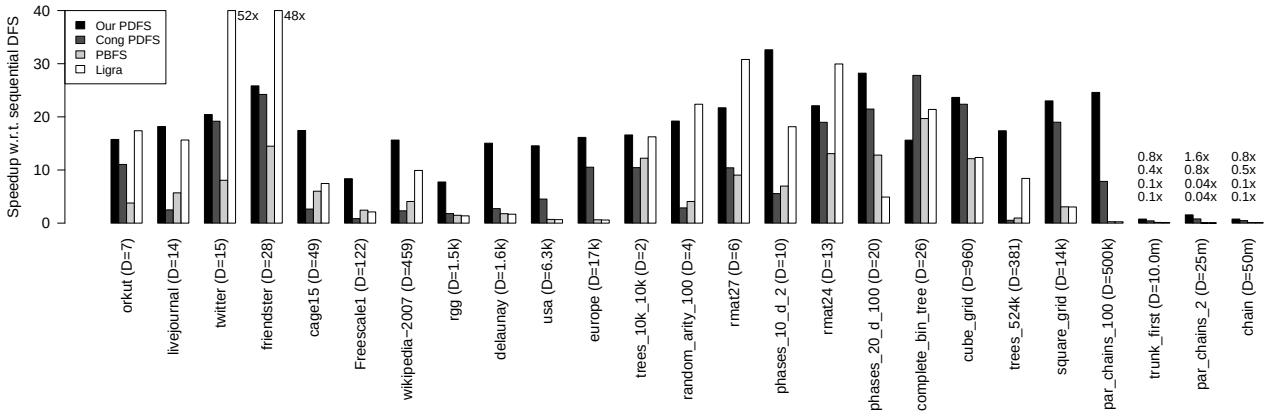

Figure 6: Speedup relative to sequential DFS. Parallel runs use 40 cores. Each graph is annotated by its diameter D.

average, which we consider to be acceptable.

The speedups achieved by our PDFS are good, overall, considering the effect of the overheads imposed by the frontier and NUMA allocation. On the real-world graphs, the speedups range from 7.7x to 25.8x and, on synthetic graphs, from 0.8x to 32.6x. Our PDFS achieves good speedups on challenge graphs, such as *par-chains-x*, where parallelism is so scarce that most other algorithms achieve no speedup at all. Moreover, our PDFS achieves excellent speedups on the grid graphs, given that the grid graphs have comparatively high diameter and expose a comparatively moderate amount of parallelism when compared to small-world graphs with a similar footprint in memory.

The throughput (expressed in million of edges processed per second) in sequential DFS varies significantly between graphs, as shown in the column labeled "seq.DFS (mEdge/s)" in Table 1. This variance has at least three explanations: (1) the measure mEdge/s does not take into account the number of vertices, although they account for work, (2) graphs with higher density benefit from better locality in the processing of the adjacency lists, and (3) graph with fewer vertices have a smaller visited array and hence benefit from fewer cache misses when accessing its cells.

6.4 Comparison with Other Parallel Algorithms

In addition to an optimized implementation of the serial DFS algorithm, which serves as our baseline, we compare our algorithm to existing state of art in parallel unordered DFS, and in parallel BFS algorithms. The comparison with the BFS algorithms is not quite an "apples-and-apples" comparison, because there are substantial differences between parallel BFS and DFS algorithms. For example, depending on the application, one algorithm may be used when the other may not be applicable. The comparison, nevertheless, gives us another data point, and can be important in applications where either an ordered DFS or BFS can be used. We describe below the algorithms compared and the implementations used.

- **Cong et al's algorithm [20].** Since there is no publicly available implementation of this algorithm, we implemented it ourselves and thoroughly optimized the implementation. We implemented batches of vertices by fast, fixed-capacity stacks, each storing 32 vertex ids (other capacities lead to worse performance). For load balancing we used the state-of-the-art concurrent-deque algorithm proposed by Chase and Lev [17].
- **The parallel breadth-first search (PBFS) algorithm of the Problem Based Benchmark Suite (PBBS) [14].** We used publicly-available sources and the Cilk Plus scheduler provided with GCC [28].
- **Ligra's direction-optimizing parallel BFS algorithm [45].** This algorithm was first proposed by Beamer [11]. We used the publicly-available Ligra sources and the Cilk Plus scheduler provided with GCC. Ligra optimizes performance for small-world graphs (e.g., social-network graphs) by switching between traversal of the out-edges of the frontier and traversal of the out-edges of the unvisited vertices depending on the relative size of the frontier. It has been shown that this technique can improve performance dramatically in small-world graphs; it can, however, lead to slowdowns in other graphs.

Figure 6 shows speedup results for our PDFS, Cong's PDFS, PBBFS PBFS, and Ligra with respect to the baseline (sequential DFS). We next discuss the most important aspects of these results.

Comparison with Cong's PDFS. Considering PDFS results (Figure 6), we observe that our algorithms outperforms Cong et al's algorithm on all graphs, except for one (*complete-binary-tree*), which is balanced and regular. On a few graphs, our PDFS slightly outperforms Cong et al's algorithm: on *twitter* (+6.5%) and *friendster* (+6.6%), and on *cube-grid* (+5.7%). On other graphs, our PDFS significantly outperforms Cong et al's algorithm by more than 15%, and by as much as a 9.8x on *Freescale1*, and a 32x on the *trees-524k* graph. We also observe that Cong et al's batching strategy induces noticeable overheads in certain graphs, such as parallel chains. Overall, we attribute our higher speedups (1) to our ability to exploit parallelism at the edge level, where Cong et al do not, and (2) to our load balancing operations that can transfer half of the frontier—not just a small constant number of vertices, and (3) to our algorithm's ability to limit scheduling overheads by using our amortization techniques.

Comparison with PBFS. Comparing the speedups of PDFS vs PBFS, we observe that our algorithm is faster in all but one case. The only case where PBFS performs better is for the perfect binary tree graph, for which synchronizing all the processors at each of the $\log n$ phases actually helps PBFS achieve a close-to-optimal load balancing in this specific situation. At the other end of the spectrum, on the *par-chains-100* graph, in which every BFS frontier

stores exactly 100 vertices (not enough to take advantage of parallelization), PBFS runs in 67s, slower than the DFS baseline which runs in 17.95s, whereas our PDFS runs in 0.73s, thus exhibiting a speedup of 24.6x. Overall, our PDFS algorithm runs 91x faster than PBFS on this worst-case graph. In general, even though results are not always as extreme, our results confirm that, as expected, PDFS typically outperforms PBFS significantly on large diameter graphs.

Comparison with Ligra. Our PDFS delivers performance that is either comparable to or better than Ligra's in all but three graphs. The benefits of PDFS are most clearly visible on large diameter graphs, e.g., in road-network graphs PDFS is 22x faster that Ligra. On the *twitter* and *friendster* graphs, Ligra achieves superlinear speedups (above 40x) with respect to the sequential DFS baseline. On these graphs, Ligra is able to avoid processing all the edges, whereas the DFS baseline and our PDFS algorithm process every edge. Ligra also outperforms our algorithm on the *rmat* graphs. On these three graphs, which Ligra specifically targets, Ligra has a 2x advantage over our PDFS. This comparison shows that our PDFS significantly outperforms Ligra in large-diameter graphs and remains competitive with Ligra in most other cases. The only exceptions are a few graphs with very small diameter, where Ligra holds the advantage. (Remark: the Ligra paper [45] reports results on the *twitter*, *rmat24*, and *rmat27* graphs; the speedups reported there differ from our results because the Ligra paper uses a direction-optimizing BFS as baseline, whereas we use sequential DFS.)

6.5 Exploiting locality

Graph-search algorithms are broadly used in garbage collectors to identify reachable memory objects. Research on garbage collection shows that DFS outperforms BFS (e.g., [29, 27]), partly because allocators typically allocate parent and child objects side by side in the heap, which gives DFS better locality. Some parallel scheduling techniques, such as the work-stealing technique used by our PDFS algorithm, have been shown to approximate the locality of sequential algorithms well partly because they minimize the number of computation migrations [4]. To determine whether our PDFS algorithm can take advantage of the locality exhibited by the serial DFS algorithm, we perform the following experiment: for each graph, we relabel (offline) the out-edges of each vertex to occur in the same order in which they are visited by our sequential DFS. We then measure the improvement in performance due to this relabeling, both for sequential and parallel DFS. Figure 8 shows the results in terms of speedup with respect to the original layout. This relabeling always improves performance, sometimes dramatically. For the real-world graphs, performance increase from 20% to 3.6x. For synthetic graphs, we even observe improvements that are as high as 45x. The self-relative improvement is, for all real-world graphs but one (*Freescale1*), relatively similar for parallel DFS and for sequential DFS. This shows that the parallelization of DFS essentially preserves the benefits of locality. These result suggests that our PDFS algorithm may indeed benefit, like sequential DFS, from better intra-graph locality.

7 Related Work

We discussed the most closely related work earlier in the paper. In this section, we discuss a number of other related work.

Work-efficient BFS. Leiserson and Shardl present a work-efficient algorithm for parallel BFS [32]. Their approach is based on a splittable bag data structure that can be used to represent the vertices in the frontier. In terms of operations supported, the main difference between our data structure and the bag data structure is that our frontier data structure supports edge-weighted balanced split operations, whereas the bag data structure supports approximately balanced splits in terms of the number of vertices. In terms of the internal structure, the data structures are quite different: we rely on a bootstrapping techniques whereas the bag data structure is more like a binomial tree. As part of our evaluation, we considered the Leiserson and Schardl implementation [32] but decided not to present these results here, because, for the graphs considered here, it was slower than the PBFS algorithm used in our evaluation; this finding is consistent with earlier ones [45].

Concurrent steal-half work queues. Hendler and Shavit propose a concurrent data structure that supports constant-time push and pop along with logarithmic-time split [26]. Our work shows that steal half using private work queues is also a viable approach. Moreover, by relying on private rather than concurrent access, we are free to use a queue structure, such as the one described in prior work [7], that offers low constant factors and asymptotically efficient operations, both in time and space. Moreover, the concurrent steal-half algorithm does not ensure that splits are amortized over sufficient work, and, as such, concurrent steal half faces the granularity-control challenges that were described in Section 2.

Hybrid algorithms. Recent work has shown benefits of using combinations of different traversal strategies. The KLA graph-processing system features a traversal algorithm that switches adaptively between PBFS (level synchronous) and PDFS (asynchronous) traversals to accelerate certain graph algorithms, such as PageRank and k-core decomposition [25]. Beamer et al [11] and subsequently Shun and Blelloch [45] propose using direction-optimizing BFS for applications, such as graph search, PageRank, connected components, radii estimation, etc.

Parallel garbage collection. In Chapter 14 of their book, Jones et al survey a number of studies of parallel garbage collection [29]. The survey identifies three mark-sweep collectors that use PDFS during the mark phase. To tame overheads, the algorithms proposed by Endo et al [22] and Siebert [47] rely on batching schemes that bear resemblance to the batching scheme proposed by Cong et al. The algorithm proposed by Flood et al [23] uses concurrent per-worker deques. Each of these algorithms relies on sharing work at the level of vertices rather than at the level of edges. In particular, Flood's algorithm relies on sharing vertices one at a time, whereas the others share half of what is locally available at a time. However, unlike our PDFS, the ones that share half do not ensure that splits are amortized over enough work. As such, these algorithms face the granularity-control challenges that were described in Section 2.

Architecture-specific optimizations. Recent research proposes certain optimizations for improving performance of graph traversals on multicore platforms. The first class of optimizations seek to hide some of the latency of cache misses: Cher et al [18] present a prefetching technique for accelerating the mark phase of a parallel mark-sweep garbage collector, and Chhugani et al [19] use prefetching to accelerate their parallel BFS. The second class of optimizations concerns the efficiency of tracking which vertices have been visited already: Chhugani et al [19] exploit certain properties of the Nehalem architecture to eliminate the need for atomic operations. The third class of optimizations use locality-aware scheduling to accelerate PBFS traversals on machines with non-uniform memory [19, 35]. These architecture-specific optimizations are largely orthogonal to our frontier representation and amortization techniques. On the one hand, each such optimization can be viewed as a particular improvement on our algorithms. On the other hand, none of these optimizations address the main algorithmic challenges identified in Section 2.

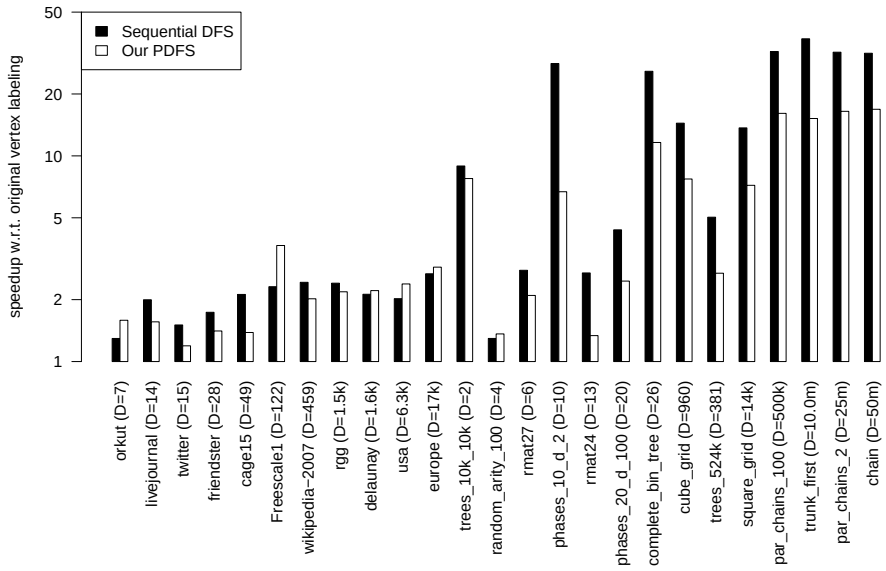

Figure 8: Speedup obtained when considering graphs with vertices relabeled in DFS order, compared with the performance of the same algorithm on the original graph. Parallel runs use 40 cores. Each graph is annotated by its diameter D.

8 Conclusion

We presented a provably work-efficient parallel algorithm for unordered DFS that delivers good practical performance. The techniques behind the algorithm include bounding the overheads of scheduling, such as thread-creation and load balancing, by amortization and by using a novel data structure for representing frontier sets. The data structure enables fast operations at both the level of vertices and edges by using a hierarchical representation and supports balanced split operations to create parallelism as needed. Our empirical evaluation shows that the algorithm performs well for a wide range of graphs including graphs with high diameter, and graphs with relatively little parallelism. The algorithm is also able to take advantage of the natural locality in certain graph layouts.

9 Acknowledgments

We thank Julian Shun for his help in using Ligra sources and producing Ligra results reported in our experimental evaluation. This research is partially supported by the European Research Council under grant number ERC-2012-StG-308246 , and by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940.

10 References

[1] Stanford large network dataset collection. http://snap.stanford.edu/.

[2] The 9^{th} dimacs implementation challenge, 2013. http://www.dis.uniroma1.it/challenge9/.

[3] The 10^{th} dimacs implementation challenge, 2014. http://www.cc.gatech.edu/dimacs10/.

[4] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems (TOCS)*, 35(3):321–347, 2002.

[5] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[6] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13*, 2013.

[7] U. A. Acar, A. Charguéraud, and M. Rainey. Theory and practice of chunked sequences. In *ESA 2014*, volume 8737 of *LNCS*, pages 25–36. Springer Berlin Heidelberg, 2014.

[8] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pages 1–11, 2010.

[9] A. Aggarwal, R. J. Anderson, and M. Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990.

[10] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray MTA-2. In *2006 International Conference on Parallel Processing (ICPP 2006), 14-18 August 2006, Columbus, Ohio, USA*, pages 523–530, 2006.

[11] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *SC '12*, pages 12:1–12:10, 2012.

[12] R. Berrendorf and M. Makulla. Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems. In *FC '14*, pages 26–31, 2014.

[13] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Room synchronizations. In *SPAA '01*, pages 122–133. ACM, 2001.

[14] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP '12*, pages 181–192, 2012.

[15] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, Sept. 1999.

[16] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM SDM*, 2004.

[17] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05*, pages 21–28, 2005.

[18] C.-Y. Cher, A. L. Hosking, and T. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *ASPLOS '04*, volume 38, pages 199–210. ACM, 2004.

[19] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *IPDPS '12*, pages 378–389. IEEE, 2012.

[20] G. Cong, S. B. Kodali, S. Krishnamoorthy, D. Lea, V. A. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

[21] T. A. Davis. University of florida sparse matrix collection, 2010. Available at http://www.cise.ufl.edu/research/sparse/matrices/.

[22] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *SC '97*, pages 48–48. IEEE, 1997.

[23] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM '01*, 2001.

[24] T. Hagerup. Planar depth-first search in o(log n) parallel time. *SIAM J. Comput.*, 19(4):678–704, 1990.

[25] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *PACT '14*, pages 27–38, New York, NY, USA, 2014. ACM.

[26] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *PODC '02*, pages 280–289, 2002.

[27] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 69–80, 2004.

[28] Intel. Cilk Plus. http://www.cilkplus.org/.

[29] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[30] V. Kumar and V. Rao. Parallel depth first search. part ii. analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[31] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW '10*, pages 591–600. ACM, 2010.

[32] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm. SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[33] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM '07*, pages 29–42. ACM, 2007.

[34] D. Mizell and K. J. Maschhoff. Early experiences with large-scale cray XMT systems. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–9, 2009.

[35] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP '13*, pages 456–471. ACM, 2013.

[36] M. Patwary, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 827–835, May 2012.

[37] M. J. Quinn and N. Deo. Parallel graph algorithms. *ACM Comput. Surv.*, 16(3):319–348, 1984.

[38] V. Rao and V. Kumar. Parallel depth first search. part i. implementation. *IJPP*, 16(6):479–499, 1987.

[39] E. Reghbati and D. G. Corneil. Parallel computations in graph theory. *SIAM J. Comput.*, 7(2):230–237, 1978.

[40] E. Reghbati (Arjomandi) and D. Corneil. Parallel computations in graph theory. *SIAM JoC*, 7(2):230–237, 1978.

[41] J. H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[42] V. A. Saraswat, P. Kambadur, S. B. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In C. Cascaval and P.-C. Yew, editors, *PPOPP*, pages 201–212. ACM, 2011.

[43] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 76–85, New York, NY, USA, 2013. ACM.

[44] E. Saule and Ü. V. Çatalyürek. An early evaluation of the scalability of graph algorithms on the intel MIC architecture. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 1629–1639, 2012.

[45] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP '13*, pages 135–146, New York, NY, USA, 2013. ACM.

[46] J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 143–153, 2014.

[47] F. Siebert. Concurrent, parallel, real-time garbage-collection. In *ACM Sigplan Notices*, volume 45, pages 11–20. ACM, 2010.

[48] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS*, 36(3):10:1–10:51, Sept. 2014.

[49] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EUROSYS '09*, pages 205–218. Acm, 2009.

[50] Y. Xia and V. K. Prasanna. Topologically adaptive parallel breadth-first search on multicore processors. In *IASTED '09*, volume 668, page 91, 2009.

APPENDIX

A Proof of Splitting Lemma

We present the proof for the lemma 5.4, which bounds the total cost of the split operations.

Lemma (Maximal cost of the split operations) *The total cost of all split operations in a PDFS execution is* $O(\frac{mB\log_B(4K)}{K})$.

PROOF. The cost of splitting a frontier of size f is $O(B\log_B f)$. Thus, there exists a constant r such that the cost of a split is bounded by "$r\,B\log_B f$". (We here rely on the fact $\log_B f > 0$, which we know because $f > 1$ when a split is executed.) For the purpose of the proof, we introduce three constants: $a = 2\log_B K + 6\log_B 2$, and $b = 4\log_B 2$, and $c = \frac{3rB}{K}\log_B(4K)$. In what follows, we associate c units of potential with every edge. The potential associated with an edge is gained by the processor that discovers the source vertex associated with this edge. We show that, globally, the potential gained by all processors suffices to pay for all the split operations that they perform. Note that the total potential is thus equal to cm, which is $O(\frac{mB\log_B(4K)}{K})$, matching the bound claimed.

To keep track of the potential gained but not yet spent, we introduce a potential function, defined as the sum of the potential of all processors. We write $\Phi(\mathtt{nb}, f)$ the potential of a processor with a frontier of size f and with $\mathtt{nb}$ denoting the number of edges processed since the last split. We define the potential as follows:

$$\begin{array}{lcl}\Phi(\mathtt{nb}, f) & \equiv & \Phi_1(\mathtt{nb}, f) + \Phi_2(f)\\ \Phi_1(\mathtt{nb}, f) & \equiv & \frac{rB}{K}\log_B K \cdot (\mathtt{nb} + f)\\ \Phi_2(f) & \equiv & \mathsf{if}\ f \le \frac{K}{2}\ \mathsf{then}\ 0\ \mathsf{else}\ rB(\frac{a}{K}f - \log_B f - b).\end{array}$$

Note that the constant a has been defined in such a way as to make Φ_2 continuous: for $f = \frac{K}{2}$, the value of $\frac{a}{K}f - \log_B f - b$ is zero. Indeed, $\frac{a}{K}\frac{K}{2} - \log_B\frac{K}{2} - b = (\log_B K + 3\log_B 2) - (\log_B K - \log_B 2) - 4\log_B 2 = 0$. Note also that Φ_1 and Φ_2 are nondecreasing with f. For, Φ_1, this property is trivial. For Φ_2, we prove that the derivative of $\frac{a}{K}f - \log_B f$ is nonnegative when $f \ge \frac{K}{2}$. This derivative is equal to: $\frac{a}{K} - \frac{1}{f\ln B}$, which is an increasing function of f. To prove its value nonnegative on $f \ge \frac{K}{2}$, it suffices to show that its value is nonnegative for $f = \frac{K}{2}$. Indeed, we have: $\frac{a}{K} - \frac{2}{K\ln B} = \frac{2\ln K + 6\ln 2}{K\ln B} - \frac{2}{K\ln B} \ge 0$, because $6\ln 2 \ge 2$.

It remains to prove that the potential evolves as expected on every possible transition. $\diamondsuit$ First, we need to prove that the initial potential is zero. It is immediate to check that $\Phi(0,0) = 0$. $\diamondsuit$ Second, we consider the increase in potential when a processor discovers a vertex. For each edge outgoing from the new vertex, we prove that the c units of potential associated with the edge exceed the increase in potential, that is: $\Phi(\mathtt{nb}, f) + c \ge \Phi(\mathtt{nb}, f+1)$. We have $\Phi_1(\mathtt{nb}, f+1) - \Phi_1(\mathtt{nb}, f) = \frac{rB}{K}\log_B K$. Besides, we have $\Phi_2(f+1) - \Phi_2(f) \le \frac{arB}{K}$, using the fact that Φ_2 is continuous at $\frac{K}{2}$ and that $-\log_B f$ decreases with f. It follows that $\Phi(\mathtt{nb}, f+1) - \Phi(\mathtt{nb}, f) \le \frac{rB}{K}\log_B K + \frac{arB}{K} = \frac{rB}{K}(\log_B K + 2\log_B K + 6\log_B 2) = \frac{3rB}{K}(\log_B K + \log_B 4) = c$, as required. $\diamondsuit$ Third, we prove that when a processor treats an edge, the potential does not increase, that is: $\Phi(\mathtt{nb}, f) \ge \Phi(\mathtt{nb}+1, f-1)$. This result follows from $\Phi_1(\mathtt{nb}+1, f-1) = \Phi_1(\mathtt{nb}, f)$ and Φ_2 nondecreasing with f. $\diamondsuit$ Fourth and last, we consider the case of a split, which involves two processors: one with a frontier of size f, and another with an empty frontier. We need to show that the total potential after the operation is no more than the potential before the operation minus the cost of the split, which is bounded by $rB\log_B f$. Technically, we need to prove: $\Phi(\mathtt{nb}, f) + 0 \ge rB\log_B f + \Phi(0, \lceil\frac{f}{2}\rceil) + \Phi(0, \lfloor\frac{f}{2}\rfloor)$. (Recall that the values of $\mathtt{nb}$ are reset to zero after a split, both for the sender and the receiver.) According to the boolean test in Figure 5, there are two cases to consider: either $f > K$, or $\mathtt{nb} > K$ and $1 < f \le K$.

Consider the first case: assume $f > K$. On the one hand, we have $\Phi_1(\mathtt{nb}, f) \ge \Phi_1(0, \lceil\frac{f}{2}\rceil) + \Phi_1(0, \lfloor\frac{f}{2}\rfloor)$. This result follows from $f = \lceil\frac{f}{2}\rceil + \lfloor\frac{f}{2}\rfloor$ and the fact that $\mathtt{nb}$ is always nonnegative. On the other hand, we are able to prove: $\Phi_2(f) \ge rB\log_B f + \Phi_2(\lceil\frac{f}{2}\rceil) + \Phi_2(\lfloor\frac{f}{2}\rfloor)$. Indeed, since $f > K$, we have $\lceil\frac{f}{2}\rceil \ge \lfloor\frac{f}{2}\rfloor \ge \frac{K}{2}$, and thus the inequality is equivalent to: $rB(\frac{a}{K}f - \log_B f - b) \ge rB\log_B f + rB(\frac{a}{K}\lceil\frac{f}{2}\rceil - \log_B\lceil\frac{f}{2}\rceil - b) + rB(\frac{a}{K}\lfloor\frac{f}{2}\rfloor - \log_B\lfloor\frac{f}{2}\rfloor - b)$, which, exploiting again the equality $f = \lceil\frac{f}{2}\rceil + \lfloor\frac{f}{2}\rfloor$, dividing by rB and unfolding the definition of b, simplifies to: $\log_B\lceil\frac{f}{2}\rceil + \log_B\lfloor\frac{f}{2}\rfloor \ge 2\log_B f - 4\log_B 2$. To justify this inequality, we observe that $\log_B\lceil\frac{f}{2}\rceil \ge \log_B\lfloor\frac{f}{2}\rfloor \ge \log_B\frac{f}{4}$ when $f \ge 2$ (which holds since $f > K$ and $K \ge 1$). Thus, $\log_B\lceil\frac{f}{2}\rceil + \log_B\lfloor\frac{f}{2}\rfloor \ge 2\log_B\frac{f}{4} = 2\log_B f - 4\log_B 2$, as required. In conclusion, the inequality $\Phi(\mathtt{nb}, f) \ge rB\log_B f + \Phi(0, \lceil\frac{f}{2}\rceil) + \Phi(0, \lfloor\frac{f}{2}\rfloor)$ holds under the first split condition.

Consider the second case: assume $1 < f \le K$ and $\mathtt{nb} > K$. On the one hand, we can prove $\Phi_2(f) \ge \Phi_2(\lceil\frac{f}{2}\rceil) + \Phi_2(\lfloor\frac{f}{2}\rfloor)$. Indeed, since $\lfloor\frac{f}{2}\rfloor \le \frac{K}{2}$, we have $\Phi_2(\lfloor\frac{f}{2}\rfloor) = 0$, and since Φ_2 is nondecreasing and $f \ge \lceil\frac{f}{2}\rceil$, we have $\Phi_2(f) \ge \Phi_2(\lceil\frac{f}{2}\rceil)$. On the other hand, we can prove: $\Phi_1(\mathtt{nb}, f) \ge rB\log_B f + \Phi_1(0, \lceil\frac{f}{2}\rceil) + \Phi_1(0, \lfloor\frac{f}{2}\rfloor)$. Indeed, this inequality is equivalent to $\frac{rB\log_B K}{K}(\mathtt{nb}+f) \ge rB\log_B f + \frac{rB\log_B K}{K}\lceil\frac{f}{2}\rceil + \frac{rB\log_B K}{K}\lfloor\frac{f}{2}\rfloor$, which simplifies to: $\frac{\mathtt{nb}}{K}\log_B K \ge \log_B f$. The latter follows from the assumptions $\mathtt{nb} > K$ and $K \ge f$. In conclusion, the inequality $\Phi(\mathtt{nb}, f) \ge rB\log_B f + \Phi(0, \lceil\frac{f}{2}\rceil) + \Phi(0, \lfloor\frac{f}{2}\rfloor)$ also holds under the second split condition. $\square$